Documentation Flutter

Documented by: Erfan Alizada

What is a StatelessWidget?

A class extends StatelessWidget when configuring a UI which does not change in form. In other words when creating a UI component that contains elements that always remain the same we use this object. The best way to use StatelessWidget is for creating different static custom widgets such as custom buttons, text boxes and other kind of UI elements.

What is a StatefulWidget?

A StatefulWidget is used when a UI component has to re-render the UI to display the updated UI or updated state of the Widget extending this class. As an example when having a button click which after pressing needs to display some sort of image or perform a task that can affect the display or the Widget component visuals.

When to use StatefulWidget and when to use StatelessWidget?

Think about a UI Widget that displays a dice picture. It contains a button which after clicking will role the dice and the UI shows the new dice picture on the screen.
In this case we can divide the application into two different parts, which is being as follow:

- ***SatelessWidget:*** Stateless refers to being immutable(not changeable) We can make a custom button and display widgets components that remain the same and will not change. Take the overall look of the Widget for instance where you have a column, a container for the Image and it's styles including a (custom) button.
  The necessary code for creating a class which can render a StatelessWidget is the following:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomeScreen(),
    );
  }
}
```

  - We extend StatelessWidget for our class in the first step.
  - We override the Widget build function which actually is required for a StatelessWidget. This is responsible for building our UI Widget. Therefore the Type of this function is Widget, which means we have to return a Widget for this function to make our app work.
  - MaterialApp is also a Widget which is suggested to use as default by material design guidelines. This Widget is a parent for all widgets that are going to be nested. This Widget has some properties that makes it easier to build a UI and it's navigation for instance.

- After nesting our Widgets within MaterialApp we return the MaterialApp to Widget build function so it get's executed and build which as a result we see the UI displaying on the screen.

- ***StatefulWidget:*** For this we can create the part of the UI that is responsible for updating the Widget and re-rendering it to display the changes. For instance when clicking the "role dice" button the default picture of dice should be replaced with another picture of dice displaying a different number.

  Note: *In order to make a StatefulWidget we always need to have to classes. The reason is that flutter manages the StatefulWidgets internally in this way. One of the classes is forced to extend StatefulWidget which doesn't accept a build function like StatelessWidget but instead it requires a createState() function* ***which returns a "State" with the same type as our custom widget.*** See the example below:

```
class DiceRoller extends StatefulWidget{
  const DiceRoller({super.key});

  @override
  State<DiceRoller> createState(){
  return
}
}
```

Keep in mind that ***"State<>"*** *has a generic type which means that we can assign any type to it according to our need and our code.*

On the other hand we need a class which is going to extend from State having the type of our custom Widget. This widget has to ***implement the "build" function.*** *Which is going to allow the class to build the UI widget. Afterwards we return this class in our DiceRoller class.*
This time the UI will be handling the changes as well, in opposite to the StatelessWidget. See the example below:

```
class DiceRoller extends StatefulWidget {
  const DiceRoller({super.key});

  @override
  State<DiceRoller> createState() {
    return _DiceRollerState();
  }
}

class _DiceRollerState extends State<DiceRoller> {
  @override
```

```
  Widget build(context) {
    return const Text("data");
    // Other code goes here
  }
}
```

Tip: The used "_" in the DiceRollerState() which makes it _ DiceRollerState() has a meaning in dart. ***It means that this class is private and only visible within the same file that is being created.***

- ***Why to have this separation?***

  Surly we could have had all the UI elements just in one class by just extending the *StatefulWidget* instead of *StatelessWidget*. But in this case we would have lost the benefit that we could get out of using ***"const"*** keyword which is a big deal while developing an application.
  The reason for that lays in understanding what ***"const"*** actually does.

  const: const is a keyword which can be used for variables and Widgets. In order to understand the use and benefit of this keyword we have to first compare it to some other keywords which can be used for variables and widgets as well such as ***"final"*** *and* ***"var"*** .
  The difference between ***"final", "const"*** *and* ***"var"*** is as following:
  var: Can be reassigned and you are only allowed to use it in a StatefulWidget as the state of the variable is changeable.
  final: *Can be used to disable the developer to change the state of the variable. When working with more developers it is better to use final over var to ensure that other developers don't override your variables while writing their own code.*
  const: Can be used for the same reason as final to disable developers to change the state of a variable with a *main difference that when declaring a variable or widget as const it will save this const variable or widget in the memory as it is never going to change. When needed to use this variable or widget in different part of the application then dart will reuse the const variable or widget saved in the memory instead of creating a new one like for a variable defined as final or var. This will enhance the performance of the application.*

  Conclusion:
  Separation of the *StatelessWidget and StatefulWidget* will allow us to save the *StatelessWidget* in the memory as it will never change and therefore saving it in the memory for reuse make sense and will enhance the performance of the application. It is not sensible to save the *StatefulWidgets* in the memory as we are not going to reuse them. The reason is that we expect different outcomes from this classes when being returned.

Flutter code structure:

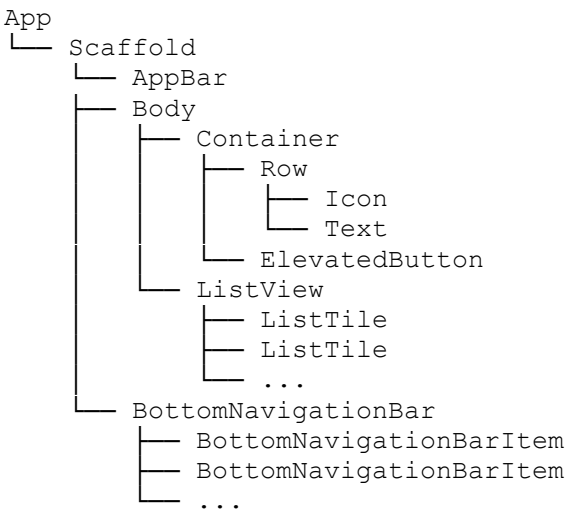
According to my experience with programming languages such as Java, Kotlin, C#, React and Python every language has its own structure and syntax. Although the core elements of programming remain the same, a programming language can vary a lot from each other in syntax and structure. For instance a developer can create composable functions using jetpack compose in Kotlin or in React a developer can create a component using jsx syntax and call them wherever it is needed to display the UI element. In Flutter we also can create so called StatelessWidgets and StatefulWidgets which in general represent the same idea of components in React or composable functions in Kotlin. But the engineering in Flutter is in a way that you can't just call your UI component and render it like that but the following points needs to be considered and needs to be understood for a successful UI Implementation and rendering.

- Each class should extend whether ***StatelessWidget*** or ***StatefulWidget***
- Depending on the type of extension you must whether implement the ***"build"*** function or not.
- Because we are extending from StatelessWidget or StatefulWidget we must pass the ***super.key*** as it is being used as a unique identifier.
- A Widget that returns the visuals for a certain screen, consist of different widgets nested in each other. In fact the whole engineering of this platform and dart language is building a UI upon nested widgets. Therefore each widget has whether a ***child or children*** property that can be set to any other widget.

Conclusion:

In order to make a good UI efficiently and professionally, it is important to have a good *nesting architecture*. Therefore before starting to code it is recommended to create a *UI Tree diagram* in which you show how your needed UI elements is supposed to be nested to achieve the desired design. You can of course use any other method that could make it easier for you, but based on my experience this makes it more understandable for me as it might for others based on its simplicity and its utility. See example on the next page(F1).

(F1)


App
Scaffold
AppBar
Body
Container
Row
Icon
Text
ElevatedButton
ListView
ListTile
ListTile
...
BottomNavigationBar
BottomNavigationBarItem
BottomNavigationBarItem
...

Findings:

1- Container() problem:

- **Problem:** Can't apply the gradient color to the background of the whole screen when using a ***Container()*** function for the body of the scaffold and setting the container's decoration to the gradient provided colors.
    - **Why?**
      Because by default the Container() gets its child's size. Which means if u have an empty container or just a Text() widget passed as the child then the color is not going to be displayed for the whole screen but just for the widget that has been passed for the child property.
        - **Solution:**
          Set the height and width of the Container() to MediaQuary.of(context).

```
-   return MaterialApp(
-         home: Scaffold(
-           body: Container(
-             width: MediaQuery.of(context).size.width,
-             height: MediaQuery.of(context).size.height,
-             decoration: returnBoxDecoration(),
-             child: const Column(
-               children: [
-
-               ],
-             ),
-           ),
-         ),
-       );
```

2- Creating a table failed

- **Problem:** I tried to create a table that has three columns and three rows.
    - **Why?**
      Because I had a different idea than what is actually going on in dart. My expectation was to create a container that has two columns and then add three rows in each column.
        - **Solution:**
          Setting up a table works different in flutter. The setting is as follow:
          - create a Container()
          - Add a column to the Container's child property
          - Pass a list of widgets to the children property of the Column()
          - Add three Row() widgets to the list of widgets
          - Pass a list of widget for each Row() widget's children property
          - Pass three widgets for each Row()

      Summary:
      By adding one column and three rows we have actually created the column as a container for the whole table. Three created rows are representing the actual rows and the other widgets within each row is actually representing a column.
      See code below:

```
@override
  Widget build(context){
      return const MaterialApp(
        home: Scaffold(
          body: Column(
            children: <Widget>[
              Expanded(child:
              Row(
                children: <Widget>[
                  Expanded(child: Text("data")),
                   Expanded(child: Text("data")),
                    Expanded(child: Text("data"))

                ],
              )
              ),
                Expanded(child:
              Row(
                children: <Widget>[
                  Expanded(child: Text("data")),
                   Expanded(child: Text("data")),
                    Expanded(child: Text("data"))

                ],
              )
```

```
            ),
              Expanded(child:
            Row(
              children: <Widget>[
                Expanded(child: Text("data")),
                 Expanded(child: Text("data")),
                  Expanded(child: Text("data"))
              ],
            )
            )
          ],
        ),
      ),
    );
}
```

Result: